USE OF CUDA FOR THE CONTINUOUS SPACE LANGUAGE MODEL

Elizabeth A. Thompson, Ph.D
Department of Engineering
Purdue University, Fort Wayne
Fort Wayne, IN USA

Timothy Anderson, Ph.D.
Human Performance Wing
Wright Patterson Air Force Base
Dayton, OH USA

***Abstract*—The training phase of the Continuous Space Language Model (CSLM) was implemented in the NVIDIA hardware/software architecture Compute Unified Device Architecture (CUDA). Implementation was accomplished using a combination of CUBLAS library routines and CUDA kernel calls on three different CUDA enabled devices of varying compute capability and a time savings over the traditional CPU approach demonstrated.**

Keywords—CUDA;CSLM;GPU; statistical signal processing; CUBLAS

I. INTRODUCTION

With the extreme computational demands for real-time graphics and gaming applications, specialized parallel hardware graphics accelerators were developed. Soon developers realized that these same graphics hardware architectures could be used to realize improvements in non-graphics applications as well [1]. With an increasing interest in developing non-graphics algorithms for graphics hardware, this field is rapidly progressing under the umbrella General Purpose Computing on Graphical Processing Units (GPGPU) [1].

A. NVIDIA Compute Unified Device Architecture

To enable flexible programming for graphics and general purpose computing, NVIDIA developed a hardware/software architecture known as the Compute Unified Device Architecture (CUDA). CUDA provides a means of accessing the Graphical Processing Unit (GPU) to issue and manage computations [2]. In data parallel applications, it provides a powerful and relatively low cost platform with a potential for significant amount of performance speedup over a traditional CPU approach. CUDA extends C or Fortran by allowing the programmer to define functions, called kernels, that when called are executed on the GPU by potentially thousands of parallel threads [3]. Therefore, there has been an explosion of interest and research in using this platform for high performance computing [4]-[9].

B. Language Models

Language models play an important role in statistical machine translation (SMT). They are responsible for expressing the probability that a translated sentence is grammatically and semantically correct without looking at the source sentence [10]. Language models use n-grams, which are word sequences consisting of *n* words extracted from a text file and used for training the model. Most of the state-of-the-art SMT systems use the n-gram back-off language models, introduced more than 20 years ago [10]. The Continuous Space Language Model (CSLM), introduced by Schwenk [10] along with an open source implementation [11], provides an alternative to the n-gram back-off model and allows "true interpolation" of the probabilities of unseen n-grams. The CSLM was successfully implemented in a large vocabulary continuous speech recognition application [12] and to statistical machine translation [13] and shown to exhibit improvement over the n-gram back-off model in modeling of the target language. Nevertheless, this performance improvement comes with extreme computational cost. The code incorporates high performance BLAS libraries to achieve fast matrix multiplications. The Intel MKL libraries can be utilized for a small fee. ALTAS, a freely available library, is another option [14]. This work investigates the use of CUDA as an alternative to these libraries to reduce the execution time of the CSLM algorithm.

II. CONTINUOUS SPACE LANGUAGE MODEL

The CSLM algorithm defined by Schwenk [10] consists of a three layer neural network: projection layer, hidden layer, and output layer, as depicted in Figure 1.

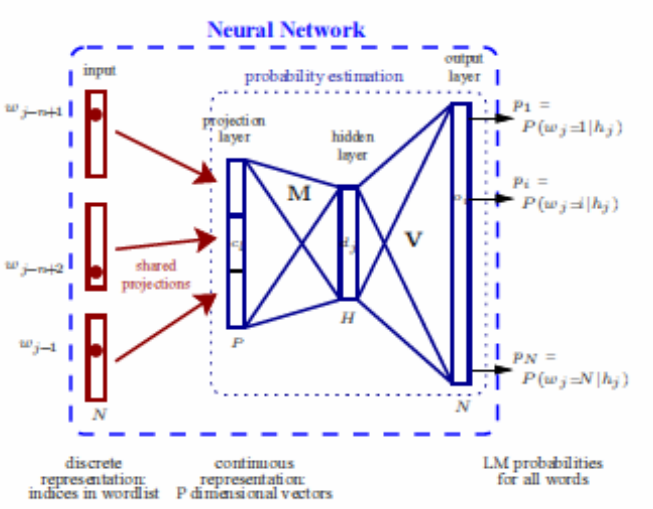

Figure 1. The CSLM architecture as designed by Schwenk

The goal is to input a 3-word sequence into the network, and the output is the probability of all words in the vocabulary being the 4th word in the sequence. The neural network must be trained through a process of adaptive learning. It is trained using 63,070 strings of 4-word sequences, known as 4-grams, obtained from a text file, news09.txt, which is also provided in the open source implementation of the software. All words from two similar files, news08.txt and news09.txt, are first combined into a text file vocab.txt, to form a list of vocabulary terms. Each of the 14,024 resulting terms in vocab.txt is assigned a numerical index, which is subsequently used in computations for training the neural network.

A. Adaptive Learning Algorithms

The basis of adaptive learning algorithms such as neural networks is to determine weights of the filter such that the mean-squared error between the filter output and the desired response is

minimum. During the adaptive learning (training) process, the filter weights are modified to move toward the minimum error solution. Propagating large blocks of data simultaneously through the network during training of the filter may speed the rate of convergence to the minimum point. However, increasing the block size may also require a reduction in the learning rate and a corresponding increase in the number of epochs to achieve the same error value, thus resulting in an increase in the overall execution time compared to a smaller block size. Thus, an optimum block size must be selected so as to ensure convergence at the fastest possible rate. Schwenk [10] defines this operation of training the network using blocks of data as bunch mode. His "out of the box" implementation uses a block size of 128 for this purpose. In his open source code, this is defined in run.csh as a command line argument to cslm_train.

B. CSLM Architecture

The CSLM is a neural network and can be considered a three stage nonlinear filter: projection layer, hidden layer, and output layer. In the training stage, values are propagated in the forward direction through the neural network from projection layer to output layer in order to assign weighting values to the input data. The filter output of stage three is compared to the numerical index of the target word, and then errors are propagated in the reverse direction, from output layer to projection layer, to improve these weighting factors. The forward and backward pass processing of all 63,070 4-grams in the training set constitutes one epoch. The error between the filter output of stage three and the target values of all 63,070 4-grams is accrued over each epoch. Training continues with additional epochs until this error has been reduced below a desired threshold or until a predetermined perplexity level has been attained. Language models are evaluated by their perplexity, which is a number which indicates the model's average branching factor, i.e., the number of words that can follow any given word.

C. Projection Layer

The projection layer accepts as input indices of the first three words of a 4-gram. The corresponding target word used at the output of stage 3 is the index of the 4th word of the 4-gram. Thus, for the 4-gram, "The Prague Stock Market," the input values to the projection layer are indices to the words "The," "Prague," and "Stock," and the corresponding target value at the output layer is the index to the word "Market." The projection layer maps each of the three input words to a unique 256 length sequence. Initially, these are generated as uniformly distributed random variables in the range -0.1 to +0.1, but their values change as the neural network is trained. For each input word, the corresponding 256 length sequence is the output of the projection layer. The projection layer thus consists of a look-up table consisting of 14024 rows and 256 columns. There are 14,024 words in the vocabulary. For each of the three input words, its corresponding index in the vocabulary list determines the row in the look-up table from which to extract the corresponding 256 values associated with that word. Thus, if the input to the neural network is a single 4-gram, the output of the projection layer is a column vector of length 768, containing the concatenation of the three 256 length sequences corresponding to the three input words. In bunch mode, 128 4-grams are simultaneously input to the projection layer. In this case, the output of the projection layer is the concatenation of three matrices, each of size 256 x 128, yielding a matrix of size 768 x 128, with each column representing the projection layer output for a single 4-gram.

D. Hidden Layer

In the forward pass, the output of the projection layer is fed as input to the second stage of the neural network—the hidden layer. The hidden layer acts as a nonlinear filter, applying weights and biases to the outputs of the projection layer. Subsequently, the hyperbolic tangent of the result is obtained. These operations are described by Eqn. 1.

$$D = \tanh(MC + B) \tag{1}$$

In Eqn. 1, C denotes the 768 x 128 output matrix of the projection layer, M is the hidden layer weight matrix of size 192 x 768, and B denotes the hidden layer bias matrix of size 192 x 128. The biases are actually in a vector of size 192 x1, which is copied 128 times to form the matrix B. The number of rows in the hidden layer weight matrix determines the output size of the hidden layer, which is selected by the architect of the network. Initially, the weights and biases are uniformly distributed random variables in the range -0.1 to +0.1, but their values change as the network is trained.

E. Output Layer

For the forward pass, the outputs of the hidden layer serve as inputs to the third layer of the neural network—the output layer. As in the hidden layer, weights and biases are applied; however, these are of different sizes than those of the hidden layer. These operations are described by Eqn. 2.

$$O = VD + K \tag{2}$$

In Eqn. 2, D denotes the hidden layer output matrix of size 192 x 128 from Eqn. 1, V is the output layer weight matrix of size 14024 x 192, and K is the output layer bias matrix of size 14024 x 128. As in the hidden layer, the weights and biases are initially uniformly distributed random variables in the range -0.1 to +0.1, but their values change as the network is trained. The outputs, O, of Eqn. 2 are subsequently subjected to softmax normalization, which is accomplished in several steps. First, the output of Eqn. 2 is applied to the computation of Eqn 3:

$$G = e^{O} \tag{3}$$

In Eqn. 3, e is the exponential function, and O is output resulting from Eqn. 2. The resulting matrix G is of size 14024 x 128, with each column representing the output for a single 4-gram. To complete the softmax normalization, the sums of each of the columns of G are calculated, and each element in matrix G is then divided by its corresponding column sum. These operations for softmax normalization are encompassed in Eqn. 4:

$$p_i = \frac{e^{O_i}}{\sum_{r=1}^{N} e^{O_r}} \tag{4}$$

In Eqn. 4, i is the column number for matrix O of Eqn. 2, and N is the number of rows in matrix O. The denominator of Eqn. 4 represents a summation over all rows in a given column of the matrix. The result is a matrix p of size 14024 x 128. Upon completion of training, p will represent the probability of each word in the vocabulary being the 4th word of the corresponding 4-gram. If a single 4-gram is input to the network, p consists of a vector of length 14024, representing the probabilities of all words in the vocabulary being the 4th word of the 4-gram. To obtain this probability, the output must be computed 63,070 times to include all possible 4-grams in the training set. In bunch mode using a

block size of 128, p is a matrix of size 14024 x 128, and each column represents the probabilities for a single 4-gram.

F. Backward Pass

After completion of the forward pass used for training the neural network, the backward pass proceeds in the reverse direction. Starting at the output layer, the outputs of matrix p of Eqn. 4 are compared to the index of the target words, and a gradient computed. Gradients are then propagated in the reverse direction in order to update the weights and biases of Eqns. 1 and 2 as well as the values in the projection layer. For specific equations used in the backward pass, the reader is referred to Schwenk's paper [10]. The forward and backward pass processing of all 63,070 4-grams in the training set constitutes one epoch. During the epoch, an error value is accumulated for all 4-grams in the training set, and an average error obtained. This is reported as the perplexity value upon completion of training.

III. USE OF CUDA FOR CSLM

The GPU is specialized for compute intensive, highly parallel computation. The CSLM algorithm is highly computationally intensive and a good candidate for implementation with CUDA. The multiplications in the hidden and output layer, both forward and backward pass, especially in bunch mode using large matrices, are highly parallel.

However, there is overhead associated with using the GPU. Memory must be allocated on both the host CPU as well as on the GPU. Variables to be used in the computation must be transferred to the GPU. The computation is then performed on the GPU, and the results must be transferred back to the host CPU.

CUBLAS is a CUDA implementation of BLAS (Basic Linear Algebra Subprogram), which performs matrix multiplication operations. It is self-contained and requires no direct interaction with the CUDA driver. Functions in the CUBLAS library provide matrix multiplications in an efficient manner and handle all overhead issues regarding programming of threads. Due to their simplicity of use, the CUBLAS libraries were used as the starting point for the introduction of CUDA to CSLM.

IV. CUDA ARCHITECTURE

The CUDA architecture consists of several streaming multiprocessors (SM), each consisting of a number of processors (a.k.a. cores) with shared memory, as shown in Fig. 2 [15]. In Fig. 2, device refers to the GPU. The number of multiprocessors and the number of cores varies, depending on the CUDA device. The CUDA programmer defines functions, called kernels. A kernel is executed as a grid of thread blocks, as shown in Fig. 3 [15], which is a group of threads that can cooperate by efficiently sharing data through fast shared memory. Each thread block is allocated to a multiprocessor, and each multiprocessor can concurrently run a maximum of 8 thread blocks. The maximum number of threads per block and threads per multiprocessor depend on the compute capability of the CUDA device and are critical parameters in the speed of execution of the code. In general, these numbers increase with increasing compute capability.

Ordinarily, the CUDA programmer defines the number of blocks per grid and the number of threads per block. The NVIDIA CUDA Software Development Kit (SDK) provides guidelines as well as numerous sample programs to assist with this. However, when using the CUBLAS functions, all these details of the kernel execution are hidden from the user. The CUBLAS programmer does not have to define kernels, grids, or thread blocks. Nevertheless, in executing code that incorporates the CUBLAS library functions, the NVIDIA Compute Visual Profiler, a graphical user interface based profiling tool, can be used to reveal the number of thread blocks and threads per block that have been defined for that code, thus providing insight into its design.

In some applications, the use of multiple GPUs can result in improved performance over that of a single GPU. These may be introduced using multiple CUDA devices or using CUDA devices containing more than one GPU.

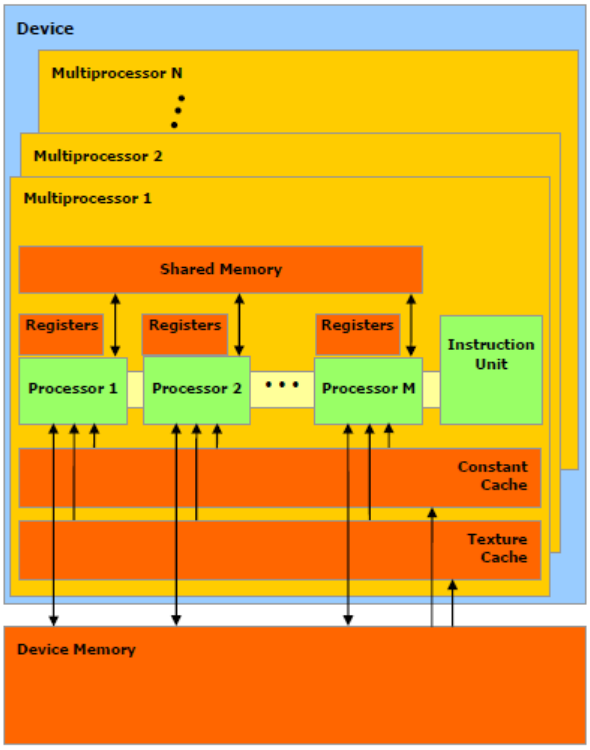

Figure 2. CUDA Architecture

V. INTEL MATH KERNEL LIBRARY (MKL)

The Intel Math Kernel Library (MKL) is a computing math library of highly optimized, extensively threaded math routines, including several core math functions such as BLAS [16]. It executes on Intel multicore CPU architectures and is an alternative to CUDA for high performance computing.

VI. METHODS

The open source implementation of Schwenk's CSLM algorithm [11] was modified to incorporate CUDA (4.0, VO.2.1221) Initially, the matrix operations in the hidden and output layer, forward and backward passes, were replaced with the CUBLAS function cublasSgemm, which performs the operation of Eqn. 5,

$$C = \alpha AB + \beta C \tag{5}$$

where A, B, and C are matrices containing single-precision values and α and β are scalars. The cublasSgemm operations replaced the BLAS gemm operations in the MachLin::Forw() and MachLin::Backw() functions of Schwenk's code. Each multiplication was handled independently, with matrices uploaded to the GPU prior to and results downloaded to the CPU after each operation.

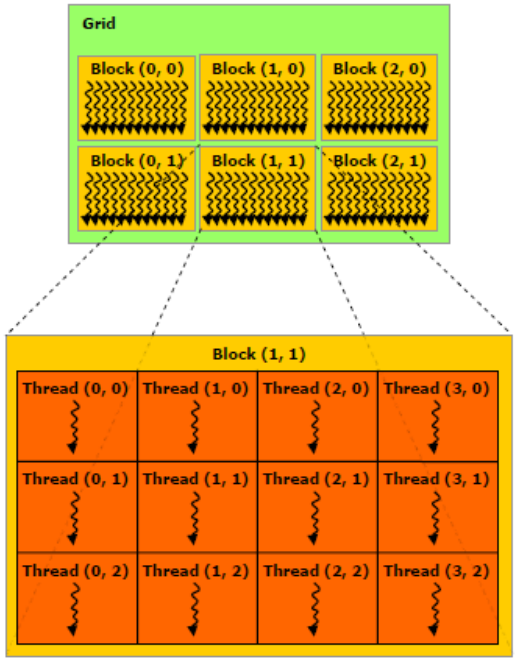

Figure 3. Grid of thread blocks

The algorithm was initially executed on a NVIDIA Quadro FX 2700M CUDA device. The details of the GPU device and the CPU platform on which it was executed are provided in Table 1. For comparison, Schwenk's original algorithm was executed on the same CPU platform using the default BLAS libraries available with Linux.

Subsequently, the algorithm was executed on two additional CUDA devices of varying compute capability, as indicated in Table 1. The GPU clock speed of all CUDA devices was between 1.2 and 1.375 GHz.

A. Intel MKL Library Implementation

In addition, the original Schwenk algorithm was executed using the Intel MKL libraries. The platform was Purdue's compute cluster, Radon, consisting of 24 8-core Dell 1950 systems with 16 GB RAM and 160 GB of disk. The processors on each of the 24 nodes consist of two 2.33 GHz Quad-core Intel E5410, for a total of 8 cores per node. The job was submitted via batch processing to a single node using all 8 cores. For comparison, the original Schwenk algorithm using the default BLAS libraries was also executed on this platform. CUDA enabled devices were not available on this platform.

With the goal of performing all operations of the CSLM algorithm on the GPU and thus minimizing GPU overhead, additional operations, such as the hyperbolic tangent operation of Eqn. 1 and the softmax normalization of Eqn. 4 were subsequently implemented on the GPU. These operations were performed with traditional kernel calls rather than the use of CUBLAS library functions. To further improve performance, redundant uploading and downloading of the matrices involved in successive multiply operations was eliminated, as described in the Results and Discussion section.

VII. RESULTS AND DISCUSSION

Schwenk's original CSLM code executed with the default BLAS libraries uses a bunch mode block size of 128 for training the neural network and 10 epochs, resulting in a perplexity value of 109.359. When this code was executed on a Hewlett-Packard (HP) laptop equipped with Duo Core Intel T9600, 2.8 GHz processors, it executed in approximately 16 minutes per epoch, resulting in a total training time of 2.62 hours for 10 epochs. On the same CPU platform, the modified code incorporating cublasSgemm functions and utilizing a Quadro FX 2700M CUDA device executed in approximated 2.5 minutes per epoch, with a total training time of 0.417 hours for 10 epochs and nearly identical perplexity value. This represents a reduction in total training time by a factor of 6.3 or 84%. These results are summarized in Table 2. Note that these results represent the first attempt rather than final results at implementing CSLM using CUDA. Additional modifications to the algorithm resulted in more savings in execution time, as discussed below.

A. Performance Comparison on Different CUDA Platforms

Subsequently, the algorithm was executed on two additional CUDA devices of varying compute capability, as indicated in Table 1. The best performance was achieved using a Quadro FX 5800 device, which has the most multiprocessors. Regardless of the compute capability, each multiprocessor can concurrently run a maximum of 8 thread blocks. Thus, the Quadro FX 5800 with its 30 multiprocessor has the largest potential number of thread blocks executing in parallel. Nevertheless, the NVIDIA Compute Visual Profiler indicated that the Quadro FX 5800 used only 24 of the 30 available SMs and 8 blocks per SM. Thus, in this implementation, the CUBLAS function cublasSgemm was perhaps not fully utilizing the potential of the GPU. This may be attributed to the fact that the CUBLAS libraries are designed as a generic tool to work on a variety of platforms and not specifically tuned to a particular algorithm or GPU device.

B. Performance Comparison MKL

For comparison of CUDA performance to the Intel Math Kernel Libraries (MKL), the original Schwenk algorithm was executed using the Intel MKL libraries and then the default BLAS libraries on the same platform. The results are displayed in Table 3. Although a comparison between the CUDA performance and MKL libraries has not been made using the same CPU platform, it appears that the CUDA implementation as executed in this study is not as efficient as that using MKL libraries. The overhead associated with using the GPUs may be the primary reason for this. For this reason, the possibility of eliminating or minimizing the number of transfers to and from the GPU was investigated.

C. Minimizing Data Transfers

It was discovered that the hidden and output layer weight matrices could be written to the GPU once for initialization.

Subsequently, no uploading or downloading of these matrices was necessary since all calculations involving them occurred solely on the GPU. It should be noted that the final version of the algorithm will generate these random values on the GPU using NVIDIA's CURAND library and thus not require this initial transfer from the CPU. Since the purpose in this study was to produce a solution identical to that of Schwenk for ease in debugging, the random values were generated on the CPU and transferred to the GPU. Similar redundancies were eliminated from other variables as well. For example, since matrix *D* of Eqn. 1 is fed as input to Eqn. 2, it does not need to be downloaded to the CPU between these consecutive operations. In addition, some of the remaining CPU operations in the algorithm were converted to CUDA kernel calls, such as the hyperbolic tangent operation of Eqn. 1 and the softmax normalization of Eqn. 4. These revisions resulted in a total reduction of 15 seconds per epoch. It is not believed that optimum performance has yet been achieved, and revisions to the algorithm are ongoing. The ultimate goal is to migrate all computations to the GPU.

D. Multiple GPUs

Since all of the CUDA devices used in this study contain only a single GPU, the use of multiple GPUs was not investigated. Nevertheless, it is not believed that a benefit would be realized by doing so. The CSLM algorithm as implemented is recursive, with each step dependent on the previous step. The limiting factor in this scenario is the maximum block size in bunch mode that can be used while still achieving efficient convergence. The ability to use large block sizes in bunch mode exploits the data parallel capabilities of and makes the algorithm more attractive for use with CUDA. From that standpoint, it appears that the larger the bunch mode block size, the better. For this reason, various bunch mode block sizes were attempted. However, increasing the bunch mode block size beyond 256 was counterproductive in that the corresponding increase in the number of epochs required to attain equivalent perplexity resulted in an increase in the overall execution time compared to the bunch mode block size of 128.

An additional consideration involves the memory limitations on the GPU device. Even if the bunch mode block size were not limited by the issue of efficient convergence, memory limitations on the GPU device would also restrict the allowable bunch mode block size of the CUDA implementation. In this case, the use of multiple GPUs could conceivably be used to enhance the performance of the CUDA version of the CSLM algorithm.

For these reasons, it is not believed that a benefit would be achieved by using multiple GPUs to realize Schwenk's recursive implementation of the CSLM algorithm. However, if an equivalent nonrecursive version could be developed, the use of multiple GPUs could conceivably be used to reduce execution time. Similarly, recursive algorithms other than the standard backpropagation method utilized in Schwenk's open source code, that are not feasible on standard CPU platforms because they require large amounts of memory, could conceivably be implemented efficiently and with reduced execution time using multiple GPUs.

Table 1. Execution time for initial CUBLAS version of CSLM algorithm on various platforms

CUDA device	Compute capability version number	Number of multi-Processors (MP)	Number of CUDA cores	Maximum threads per block	Maximum threads per MP	CPU platform	CPU operating system	Execution time per epoch (min)
Quadro FX 380 LP	1.2	2	16	512	1024	HP Z200 SFF workstation 4 Intel Core i3-530 processors 2.93 GHz	Fedora 2.6.33.3-85.fx13x86_64	3
Quadro FX 2700M	1.1	6	48	512	768	Duo core Intel T9600 2.8 GHz	Scientific Linux 6.0	2.5
Quadro FX 5800	1.3	30	240	512	1024	HP Z800 workstation 12 Intel Xeon x5660 processors 2.8 GHz	CentOS Linux 2.6.32-71.29.1e16.x86-64	1.33

E. Analysis of Performance Improvement

The forward pass matrix multiplications of Eqns. 1 and 2 account for 44.5% of the total execution time in Schwenk's original algorithm when implemented with the default BLAS libraries; the update of the gradient matrix in the backward pass represents another 22.9%, and the update of the weight matrix 22.7%. Thus, combined, these operations represent over 90% of the execution time of Schwenk's CSLM algorithm. The approach utilized in this study replaced these operations with the cublasSgemm command and handled each multiplication of the hidden and output layer, forward and backward pass, independently, with interim results being written back to the CPU, resulting in a reduction in execution time of 84%. In this CUBLAS implementation, the matrix multiplications executed in the cublasSgemm commands represented only 3.3% of the total execution time, whereas the uploading and downloading of interim results expended 16.3% of the execution time. Some of the redundancy and overhead has subsequently been eliminated. The ultimate goal is to migrate all computations to the GPU, thus eliminating all interim data transfers. This process is still under investigation. Another method to hide some of the GPU overhead may involve a hybrid technique in which GPU and CPU

operations are performed in parallel, such as that described by Barrachina et al. [17].

VIII. CONCLUSIONS

A framework has been provided to introduce CUDA to the training phase of the Continuous Space Language Model, and a time savings over the traditional CPU approach has been demonstrated. The algorithm was tested on CUDA devices of varying compute capability, and the best performance was achieved using a Quadro FX 5800 device, but the use of CUBLAS libraries perhaps did not fully utilize the capability of the CUDA device.

Due to their simplicity of use, the CUBLAS libraries provide a good starting point for the use of GPUs, providing matrix multiplications in an efficient manner while hiding all overhead issues from the user. Furthermore, successive multiplication operations using CUBLAS function calls do not require downloading of interim results to the CPU between multiplications and, in fact, should be avoided because redundant uploading and downloading of interim results adds overhead and reduces performance. CUBLAS library function calls can be mixed with traditional CUDA kernel calls within the same program. While the use of the CUBLAS function calls was efficient in performing matrix multiplication, further performance improvement is theoretically possible by utilizing all 30 of the SM available on the Quadro FX5800.

While not equivalent to the performance and capabilities of a supercomputer, CUDA provides a substantial performance improvement at relatively low cost, making high performance computing accessible to the average user. Furthermore, its availability on various platforms, such as laptops, may make it more appealing and practical than a supercomputer in some applications.

Due to the extreme memory requirements of the CSLM algorithm, it may not be feasible to avoid transferring interim multiplication results back to the CPU, but it may be possible to hide some of the GPU overhead using a hybrid approach that performs computation on the CPU and GPU in parallel.

The bulk of the time savings over the CPU version came from the cublasSgemm matrix multiplications because these operations account for 90.1% of the algorithm's execution time. Additional incremental savings to improve performance over that of the Intel MKL libraries resulted by minimizing data transfers to and from the GPU. Additional savings may be possible by replacing the backpropagation method with an alternative algorithm using multiple GPUs.

Table 2. Comparison of initial CUBLAS version vs. original Schwenk algorithm using Quadro FX 2700M

Algorithm	Block Size	Initial Learning Rate	Number of Epochs	Total training time (hrs)	Perplexity
Original Schwenk	128	0.005	10	2.62	109.359
CUBLAS	128	0.005	10	0.417	109.368

Table 3. Execution time of original Schwenk algorithm using two different libraries

Library	Execution time/epoch (min)
Default BLAS	21.72
MKL	1.05

REFERENCES

[1] V. Allada, T. Benjegerdes, B. Bode, "Performance analysis of memory transfers and GEMM subroutines on NVIDIA Tesla GPU cluster," *Proc. of the IEEE Interntl Conference on Cluster Computing and Workshops*, New Orleans, LA, pp. 1-9, 2009.

[2] J. Franco, J. Bernabé, J. Fernández, M. Acacio, "A parallel implementation of the 2D wavelet transform using CUDA," *Proc. of the 17th IEEE Euromicro International Conference on Parallel, Distributed, and Network-Based Processing*, Weimar, pp. 111-118, 2009.

[3] E. Phillips, M. Fatica, "Implementing the Himeno Benchmark with CUDA on GPU clusters," Proc. of the 24th IEEE International Symposium on Parallel and Distributed Processing (IPDPS), Atlanta, GA, pp. 1-10, 2010.

[4] Z. Du, Z. Yin, D. Bader, "A tile-based parallel Viterbi algorithm for biological sequence alignment on GPU with CUDA," *Proc. of the IEEE International Symposium on Parallel and Distributed Processing, Workshops and PhD Forum (IPDPSW)*, Atlanta, GA, pp. 1-8, 2010.

[5] W. van der Laan, A. Jalba, J. Roerdink, "Accelerating wavelet lifting on graphics hardware using CUDA," *IEEE Transactions on Parallel and Distributed Systems*, 22(1):132-146, 2011.

[6] B. Han, T. Taha, "Acceleration of spiking neural network based pattern recognition on NVIDIA graphics processors," *Applied Optics*, 49(1):B83-B91, 2010.

[7] J. Bowden, "Application of the OpenCL API for implementation of the NIPALS algorithm for principal component analysis of large data sets," *Proc. of the IEEE 6th International Conference on e-Science Workshops*, Brisbane, QLD, Australia, pp. 25-30, 2010.

[8] X. Chen, L. Gao, H de Garis, "CuParcone—a high-performance evolvable neural network model," *Proc. of the IEEE International Conference on Intelligent Computation Technology and Automation (ICICTA)*, Changsha, China, pp. 1070-1074, 2010.

[9] S. Liang, Y. Liu, C. Wang, L. Jian, "Design and evaluation of a parallel k-nearest neighbor algorithm on CUDA-enabled GPU," *Proc. of the IEEE 2nd Symposium on Web Society (SWS)*, Beijing, China, pp. 53-60, 2010.

[10] H. Schwenk, "Continuous-Space Language Models for Statistical Machine Translation," *The Prague Bulletin of Mathematical Linguistics*, 93:137-146, 2010.

[11] http://liumtools.univ-lemans.fr/

[12] H. Schwenk, "Continuous Space Language Models," Computer Speech & Language, 21:492-518, 2007.

[13] H. Schwenk, D. Dechelotte, J-L. Gauvain, "Continuous space language models for statistical machine translation," *Proc. of the COLING/ACL 2006 Main Conference Poster Sessions*, pp. 723-730, 2006.

[14] http://math-atlas.sourceforge.net/

[15] NVIDIA OpenCL Programming Guide for the CUDA Architecture, version 2.3, 8/27/2009, http://www.nvidia.com/content/cudazone/download/OpenCL/NVIDIA_OpenCL_ProgrammingGuide.pdf.

[16] http://software.intel.com/en-us/articles/intel-mkl/

[17] S. Barrachina, et al., "Evaluation and Tuning of the Level 3 CUBLAS for Graphics Processors," *Proc. of the IEEE International Symposium on Parallel and Distributed Processing (IPDPS)*, pp. 1-8, 2008.